In the liminal space

Thomas, Lynda; Boustedt, Jonas; Eckerdal, Anna; McCartney, Robert; Moström, Jan Erik; Sanders, Kate; Zander, Carol

Published in:
PESTLHE

Publication date:
2017

Citation for published version (APA):
Thomas, L., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., & Zander, C. (2017). In the liminal space: Software design as a threshold skill. PESTLHE, 12(2), 333-351.

Practice and Evidence of Scholarship of Teaching and Learning in Higher Education
Special Issue: Threshold Concepts and Conceptual Difficulty
Vol. 12, No.2 April 2017, pp. 333--351

In the liminal space: software design as a threshold skill

Lynda Thomas*
Computer Science
Aberystwyth University, UK
ltt@aber.ac.uk

Jonas Boustedt.
Faculty of Engineering and Sustainable Development
University of Gävle, Sweden
jbt@hig.se

Anna Eckerdal.
Department of Information Technology
Uppsala University, Sweden
Anna.Eckerdal@it.uu.se

Robert McCartney.
Department of Computer Science and Engineering
University of Connecticut, USA
Robert@engr.uconn.edu

Jan Erik Moström.
Department of Computing Science
Umeå University, Sweden
jem@cs.umu.se

Kate Sanders.
Mathematics and Computer Science Department
Rhode Island College, USA
ksanders@ric.edu

Carol Zander.
Computing & Software Systems
University of Washington Bothell, USA
zander@u.washington.edu

* Corresponding Author

ISSN 1750-8428 (online) www.community.dur.ac.uk/pestlhe.org.uk

Abstract

In previous work we proposed the idea of 'threshold skills' as a complement to threshold concepts. The definition of threshold concepts assumes that theoretical knowledge is paramount: gaining the understanding of particular concepts irreversibly transforms the learners. We noted, however, that mastering computing, like many disciplines, requires learning a combination of concepts and skills, and we suggested that this required further investigation.

In this paper we examine the activity of designing software as a possible example of such a threshold skill. We looked at 35 software designs collected from students nearing graduation in computing courses, and see many of the characteristics of threshold skill and also of students being in liminal space. A close examination of the students' designs leads to some useful implications for teaching this fundamental activity.

Keywords: Threshold concepts, Threshold skills, Professional education, Practice

Introduction

The theory of threshold concepts has been applied to computing by a number of authors: Zander et al. (2008), Shinners-Kennedy (2008) and Sorva (2010) for example (see Rountree and Rountree (2009) or Flanagan (2012) for more examples). These papers have identified a number of potential threshold concepts, many having to do with learning to program. However, one of the features of programming is that it requires skill as well as conceptual understanding. Moreover, when we interviewed students about learning concepts, they often discussed skill acquisition as an important (and difficult) aspect of their learning. One of our interviewees pointed the way:

> *There's just some aspects to (programming) that just seem to remain kind of mysterious to me at the programming level. Not the concept level, not the theory level, not the technology level, but at the kind of code nuts and bolts level ... I sense from our conversations that you (as a teacher) feel you have more problem in getting the concepts across …*

This student was explicitly telling us that we were looking for concepts, but that for her the concepts were not the problem, the 'doing' was.

This led us to argue that in computing, and possibly other disciplines, skills as well as concepts need to be considered as thresholds. We proposed in Sanders et al. (2012) and Thomas et al. (2012) some characteristics for such threshold skills. These were derived from threshold concepts, but manifested somewhat differently. We also noted one new characteristic - the importance of practice. This discussion was based on an empirical analysis of student interview data, which is described in detail in Boustedt et al. (2007) and Sanders et al. (2012).

To understand threshold skills, it is necessary to understand what we mean by skills. The following is the definition of *skill* from the Oxford English Dictionary Online (2016):

> *Capability of accomplishing something with precision and certainty; practical knowledge in combination with ability; cleverness, expertness. Also, an ability to perform a function, acquired or learnt with practice.*

It is the second of these meanings: *an ability to perform a function, acquired or learnt with practice* that we are using. This notion of skill aligns with Norman's description of "knowledge how", or procedural knowledge, which he describes as "…difficult or impossible to write down and difficult to teach. It is best taught by demonstration and best learned by practice." (Norman, 1990, p. 58) It has been observed (Norvig, 2001) that programming a computer well, like other skills, requires a good deal of time and practice. The key aspects of skills are that they relate to doing things, and that they are learned and improved by practice.

Given this understanding of skill, we adapted the definition of threshold concept to skills, and proposed that threshold skills are:

> **Transformative** Mastering a threshold skill transforms what students can do – and their vision of what they can do. It is empowering and, as a result, often accompanied by an increase in confidence; contrasted with threshold concepts, where mastery transforms how students see their discipline.

Integrative Once a threshold skill is attained by applying it to one task, students see other potential applications. Rather than unifying different concepts (as threshold concepts can do), a threshold skill broadens the list of tasks students can perform or enables them to perform them in a new way.

Troublesome Skills can be complex, demanding, and time-consuming to learn and maintain. They may seem alien at first (like the linear, step by step thinking required to debug a program). They may even be counter-intuitive.

Semi-irreversible Unlike threshold concepts, threshold skills degrade over time with lack of use. They do not completely go away, however. Students who have acquired a threshold skill stay transformed and know where the skill applies, but may need to review or practice.

Associated with practice It is inherent in our definition of skill that it is "attained or learnt through practice", where practice is "repeated exercise in or performance … so as to acquire or maintain proficiency." (Oxford English Dictionary Online, 2016) Without practice, skills are not acquired and when attempting to solve a problem or perform a task, this may frustrate and discourage students in an immediate way that failure to understand a concept might not.

Investigating Software design as a Threshold Skill

Software design is the phase of software development that takes a description of what is to be built and creates a detailed description of how it should be built. Designers must break a problem down into parts and describe solutions to those parts and how they will fit together using diagrams and other computing language artefacts.

In the big picture, software design as a concept is straightforward for students. As noted, the concept is to break a problem down into its components and describe how they work together. As a concept, this is understandable. Computing students do this kind of problem solving from the beginning of their curriculum so the nature of it is familiar to them. But the 'nuts and bolts' of actually doing it, the skill of designing the

software, of coming up with reasonable parts and being able to describe the action of the system is quite difficult for many students. When problems are small and not complex, most students can grasp an overview of the system and its parts. But as problems get larger and more complex, the threshold skill of designing software becomes more challenging.

As a research group, we previously investigated the abilities of graduating students in the field of software design. Computing students were asked to "design a super-alarm clock that university students could use to manage their sleep patterns" (Eckerdal et al. 2006) (Loftus et al. 2011). In Eckerdal et al. (2006), we used inductive analysis and came up with six categories for their solutions:

Nothing. The work had little or no intelligible content.

Restatement. These merely restated the problem requirements.

Skumtomte (meaning marshmallow fluff in Swedish). These added a small amount to restating the task, but upon investigation the designs were shown to have no substance.

First step. Designs included some significant work beyond the problem description.

Partial design. Designs provided an understandable description for each of the parts and an overview of the system that illustrates the relationships between the parts (although the work was incomplete or superficial).

Complete. Designs included an understandable overview with descriptions of the parts that included responsibilities of each part and an outline of the explicit communication between them.

We found few students gave 'complete solutions' or even a 'partial design.' More students gave a 'first step' to the design, but more than half of the students restated the problem, or added just a small amount to restating the task.

By examining students' designs, we are able to see both *skill* – what they *do* when asked to design -- and something of what they understand of the *concept* of design. Software design seemed like a likely possibility for a Threshold Skill, and in this paper we discuss this from a further examination of what students say about design in interviews, and also from examining, in detail, student software designs.

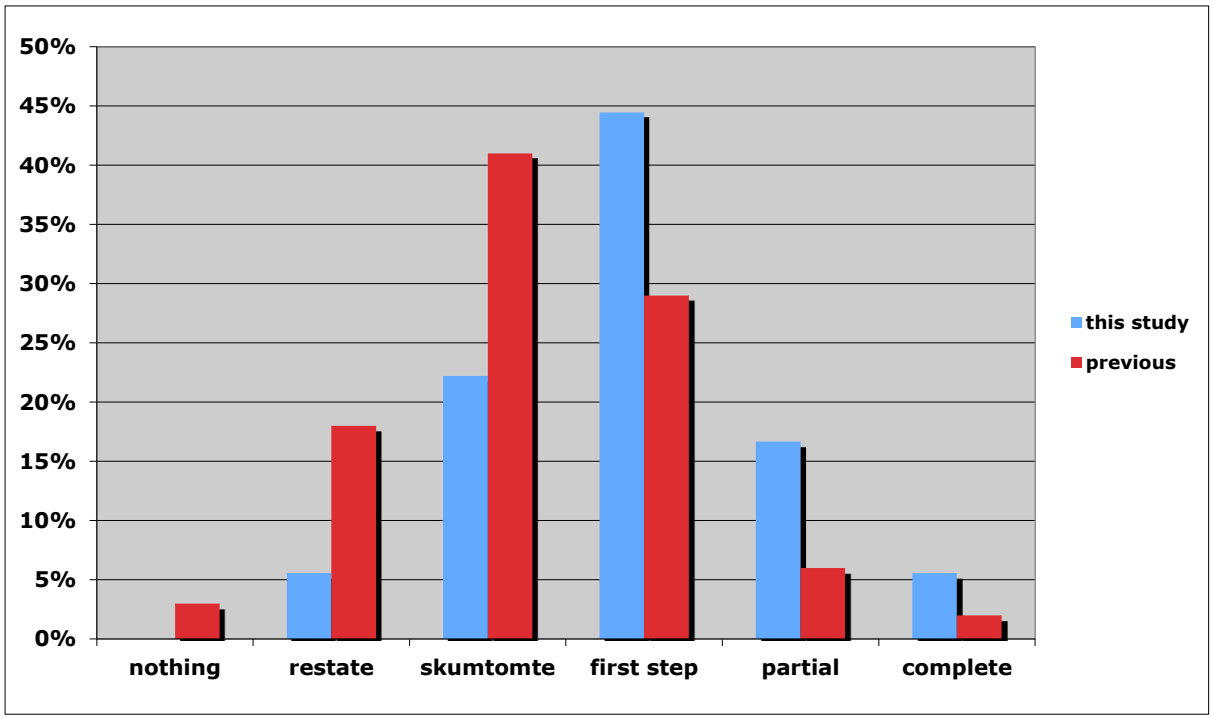
Methodology

We used two sets of data. The first set consists of early interviews asking students about threshold concepts (see Boustedt et al. (2007) and Sanders et al. (2012) for details). We re-examined the interviews looking for comments on software design and then sought out data on skills rather than concepts. As researchers, we considered the interview comments individually and then as a group.

For the more recent data set, we asked a group of graduating computing students to design the 'super-alarm clock' problem, to produce a design that someone else could work from. We collected the designs from 35 students who were taking a compulsory year-long final year projects course. They had no warning of this exercise and although they had an intensive education in computing, including software design courses, they had no opportunity to prepare or review their skills.

We then analysed the designs individually and in groups. Using software engineering concepts and influenced by Loftus et al. (2011), we created a spreadsheet that corresponded to the categories: analysis, static design, dynamic design, and linkage. After analysing the data along these lines, we compared the designs to one produced by an expert, and then categorized all the designs with respect to understandings of the design process that were exhibited. As a measure of overall quality we assigned each design into one of the six categories outlined in Eckerdal et al. (2006), and restated above. Figure 1 shows a comparison of results between this and the previous study, for more detail see Thomas et al. (2014). A greater proportion gave designs of the top three categories than had previously. We attribute the greater success of the students to a fairly uniform and more thorough preparation.

Figure 1. Comparison of previous and current results – Eckerdal et al. 2006 vs. Thomas et al. 2014



In addition, in a time-consuming iterative process of reading and comparing the designs, individual, decontextualized features were grouped and re-grouped until consensus was reached on the critical aspects that differentiated the various ways students understood the phenomenon of 'produce a design' (Thomas et al. (2014).)

Characteristics of Threshold Skill

We looked at each of the threshold skill characteristics outlined above, specifically in relation to software design.

Transformative. This was investigated through examination of the interviews rather than the designs. As one interviewee said "I now find myself looking at everyday things and working out in my head how I would represent them … I will be queuing up at the supermarket and am able to picture in my head how I would represent a checkout system using object (oriented design). It has revolutionised the way I think about computing"

Integrative. Producing a software design provides an opportunity for students to demonstrate their analytic skills and their capacity for abstraction, both of which are useful throughout computer science. One interviewee discusses seeing "the pattern" and how he can "go home and figure it out if there's a pattern to it."

Troublesome. Students certainly find software design difficult. They must learn new notation, diagrams, and terms AND problem solve. In the designs we examined, most students missed some critical part, failed to link parts, or failed to use correct professional language. An example can be seen in Figure 2, where a student made little progress with the task: doodling and restating the problem. Figure 3, on the other hand, shows a student beginning to use correct professional notation.

Semi-irreversible. The students had certainly been taught the notations and skills that were expected in this exercise; but, if they did not exhibit them we cannot be sure exactly why. Had they ever really mastered these skills, or had they forgotten them? Most students seemed to recognise that a problem should be broken into parts (so that skill may be irreversible), some were able to design the partial solutions, but few were able to integrate these solutions back together to solve the whole problem.

Must be practiced. One interviewee acknowledges the importance of practice: "It took a lot of just practicing and just repeating. It's to the point where when you see it you wouldn't be kind of intimidated. You would already say, okay I know what I can do with this."

Liminal Space

It appears that software design does have the characteristics we previously identified for a threshold skill. In addition, we noted some other characteristics of students in the liminal space, namely: partial understanding, mimicry, and a failure to recognize the boundaries of the field (McCartney et al. 2009).

Figure 2. Before the Threshold Skill

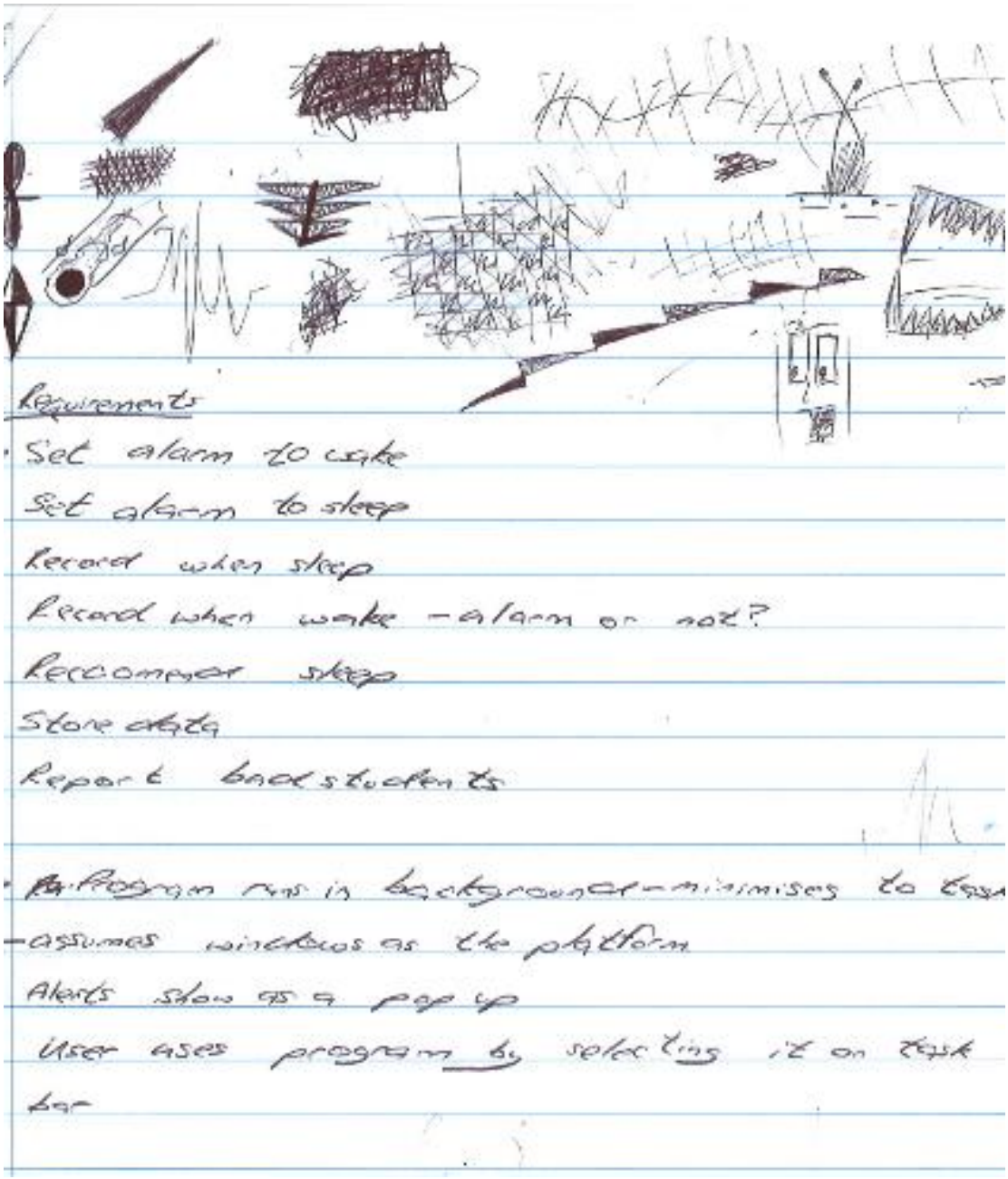
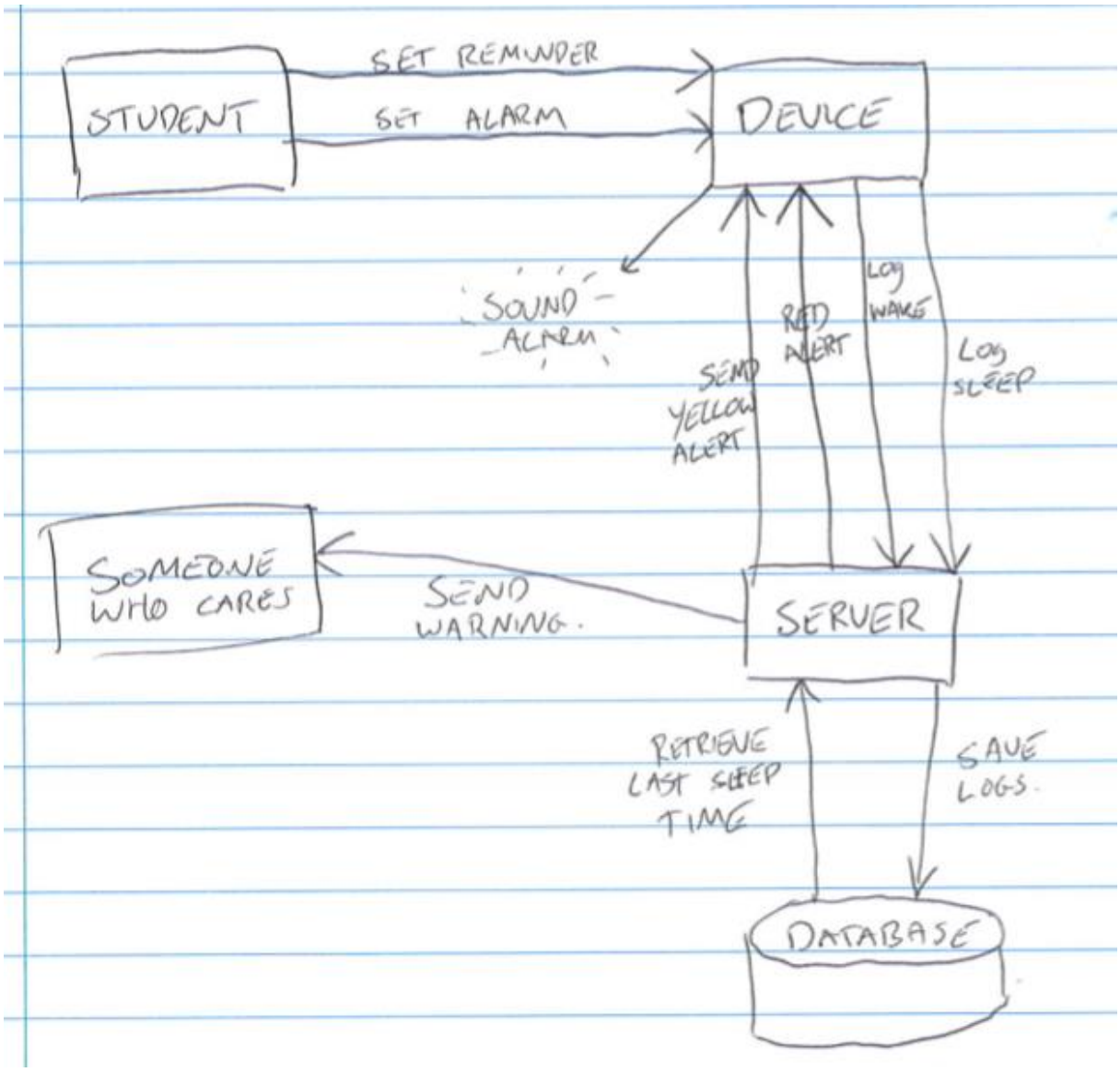


Figure 3. After the Threshold Skill



Partial Understanding (Partial Mastery)

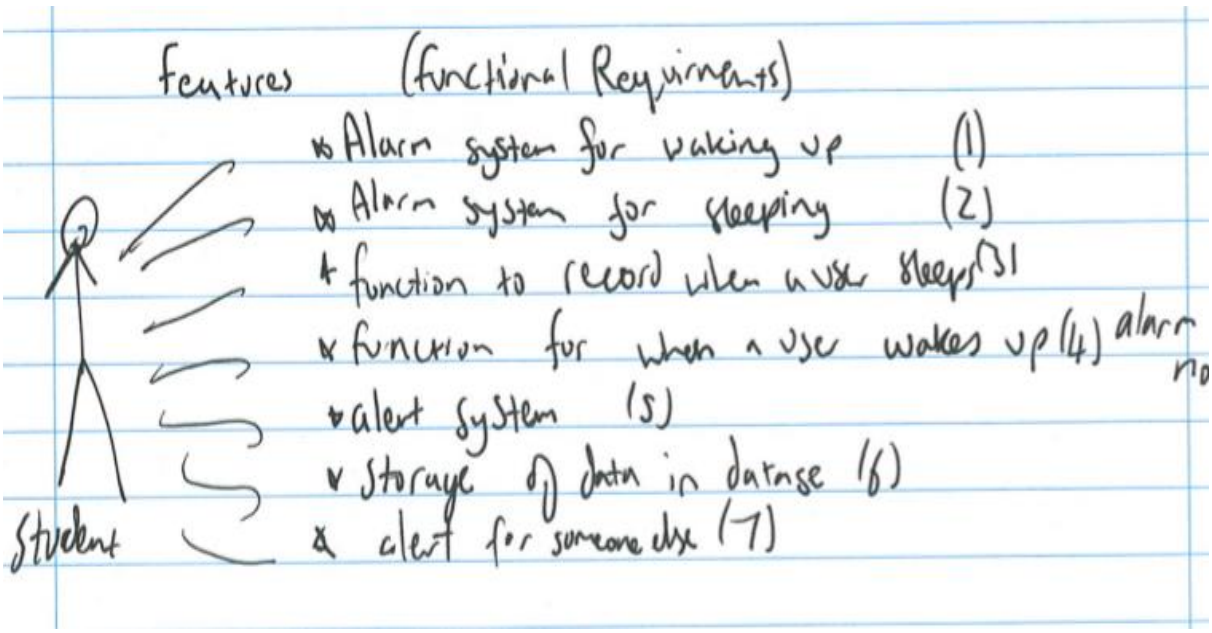
When examining threshold concepts, we saw partial understanding. Students reported instances where they believed that they could not explain a concept in the abstract, but could nevertheless apply that concept at a concrete level. Sometimes the reverse was true, that students had a theoretical understanding but could not apply it to a concrete problem (Thomas et al. (2012).

In the software designs we saw partial 'mastery' rather than partial understanding. At the weakest level, we saw students who did not demonstrate that a design involved parts. Some students alluded to parts, but failed to find solutions for the parts. Some students who identified parts to the problem, then failed to link the solutions to those parts back together. And many students failed to use the Language of Computing correctly. They had been taught formal diagrams and notation to communicate their software design but either used it incorrectly or failed to use it at all. In this sense they showed that they could perform some aspects of the desired skill but not others, akin to being in a liminal space while acquiring the skill of software design.

Mimicry

Although students did not use the language of computing correctly, we found evidence in the data of mimicry of that language. Doing a software design can involve using certain commonly accepted artefacts in the analysis and solution to the problem. One of the first artefacts that is often taught is 'use case diagrams' where the actors in the problem are represented by stick figures and other symbols. The super-alarm clock problem involves several actors. Some students drew a stick figure, but only one. We interpret this as the students knew they were supposed to draw stick figures, but did not fully master how these are used in the analysis. They rather mimicked the way they believed a design should be done. Figure 4 below shows such an example. Here, the problem is merely restated, and a stick figure added, which does not add anything to the solution.

Figure 4. A restatement showing mimicry

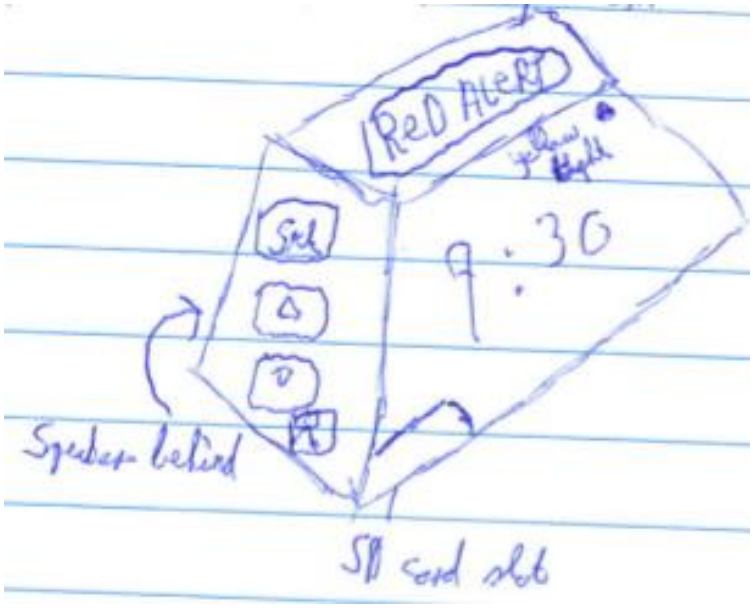


Mimicry is not necessarily something bad. When learning a skill like software design, mimicry can be a constructive, and even necessary, way to start. Some of the best designs in the second study included most of the artefacts the students had learnt in their education. These good designs used the artefacts in the intended way: as a help in the analysis and solution to the problem. The students had followed, and maybe to some extent mimicked, what the teachers had taught them. Still, what was striking was that the expert in the study, in contrast to the students, chose to use only one artefact – but did it in a way that clearly and elegantly showed a solution to the problem and that subsumed other artefacts. Box (2009) describes how experts of design may come to a point where they have a loose relation to artefacts and processes. She describes their approach as “a strategy of adapting and scaling a method with the intent to accurately define the problem while sharing a vision of the project”. These experts do not slavishly use the artefacts for the sake of a rule or convention, but rather adapt the method to the problem. This requires however a good mastery of all parts involved in the design process, how they relate to each other and how and where in the process they belong. This is far beyond the level of the fragmented ability exhibited by many of the students in our study. Mimicry can be a successful means for our students to become more skilful in design, and eventually reach the holistic view characteristic of experts, but it is not enough.

Design as a Boundary Marker

Some students produced what we would not consider software designs at all - failing to observe the boundaries of the field. The design in Figure 5 is something that we might expect from someone outside the field of computing. This is a rather sad result of 3 years of study in a field.

Figure 5. Outside the boundary



Threshold skills and students' understandings of software design

In Thomas et al. (2014) we used a phenomenographic approach on the 35 designs to come up with an outcome space with six understandings. For each category of understanding demonstrated, the student produces a richer design and is closer to mastery of the skill of designing software. As a student is going through the process of acquiring the skill, transitioning from layman to expert, the student is experiencing the liminal space. The categories were:

0. The design a layman might produce

1. A design with some formal notation

2. A design that uses formal notations to express the static relationships among the parts

3. A design that uses formal notation to express sequential (dynamic) information, but does not relate that to the static system parts

4. A design that includes and relates multiple artefacts, both static and dynamic

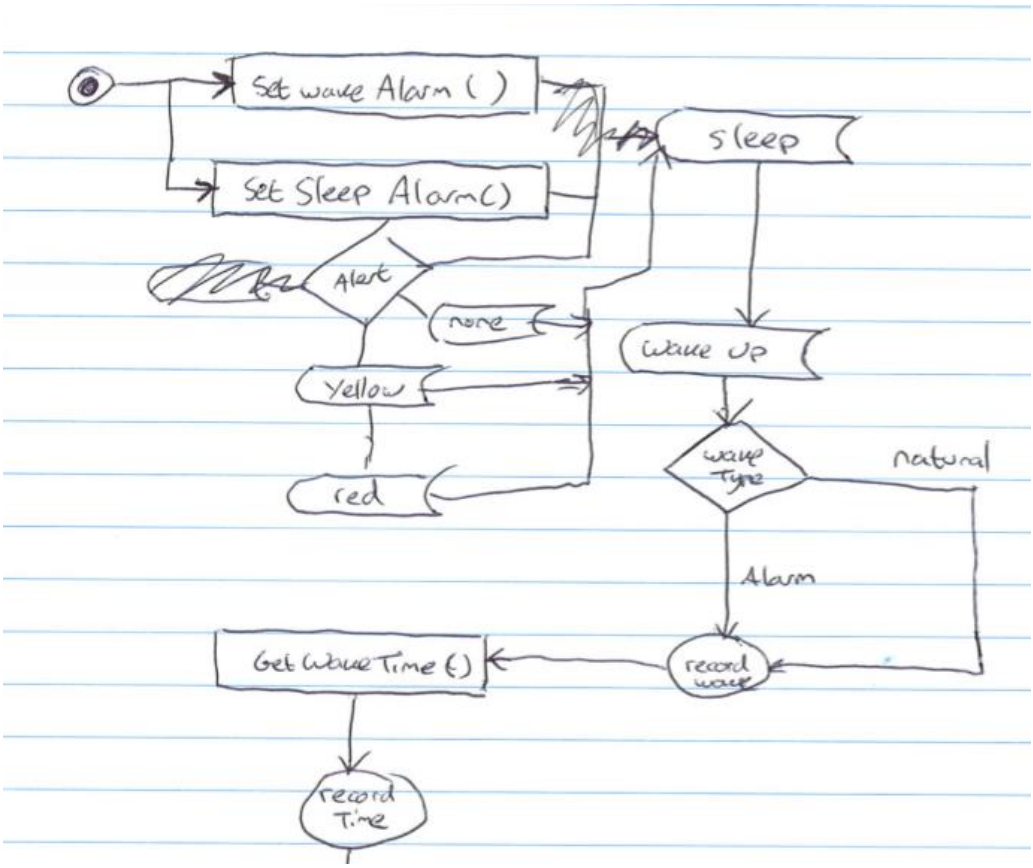
5. A design that relaxes the notations and includes only essential artefacts (this understanding was demonstrated by an expert in the discipline and is not expected of undergraduate students)

Students producing artefacts that fall into category 0 or 1 exhibit little software design skill. Some students produced designs that a layman might, such as the alarm clock in Figure 5. Failing to observe the boundaries of the field, this student is far from the threshold of designing software. The student who produced only a use-case diagram, such as that in Figure 4, shows that they know there is a language of computing by using some formal notation, but demonstrates little skill - essentially only restating the problem characteristics.

Students producing Category 2 artefacts understand that design is about software. They not only understand the concept that software has parts, but can demonstrate the skill by using class or interaction diagrams to show the parts and their relationship to each other. Category 3 artefacts take design a step further by showing some behaviour, but the software components have not been clearly identified. Figure 6 shows an example of system behaviour that does not, however, refer to the static components of the system.

Understanding that a software system has parts is well-known for most undergraduate students, but demonstrating how those parts work together is difficult. Students know that software performs tasks, but relating the tasks, the dynamic nature of the system, to the static parts is a challenging skill not easily acquired. Demonstrating Category 4 takes the skill of defining components while at the same time understanding how they interact. The diagram of Figure 3 shows a Category 4 artefact.

Figure 6. Shows behaviour, dynamic information



We do not expect students to demonstrate Category 5. This is the domain of experts. Once computing professionals have been working in the discipline for a significant period of time, they acquire the understanding and skill to relax the formal notation. They include both static (the parts) and dynamic notations (behaviour), but not always as is taught formally in school. Yet their design is clear. They are able to succinctly demonstrate good software design. This threshold skill comes with years of practice. The computing professional has emerged from the liminal space an expert.

Implications for Teaching

We sought to elicit critical aspects that differentiated the various ways students understood the instruction ‘produce a design’ in Thomas et al. (2014). In the designs we were able to identify the following critical aspects and their implications for teaching. Each critical aspect corresponds to acquiring and refining a skill:

1. There is a Language of Computing that designs should use so that they can be used to communicate between professionals. Some students have not discerned that there is a special, formal language of computing. It is important to note the differences between a layman's sense of design and a *software* design. Designs for the same problem should be created using more and more artefacts of the language of computing. By contrasting these designs, students may discern and see the purpose of the language.

2. Systems are made up of components. These components form the static structure of the system. This is a common feature of problem solving and examples that students are already familiar with can be discussed and compared with the use of general problem solving skills like divide and conquer.

3. There has to be a way of indicating behaviour in a system. This seems to be a crucial and difficult step for many students, particularly when they attempt to express it in formal notation. Students need to be challenged on their designs – asked how does this part actually work.

4. The static structures and the dynamic behaviour have to be integrated. This is where we recognise that, as instructors, we need to emphasise why the use-case diagram (Figure 4) is useful, and how it is not enough on its own. How each of the cases is going to be solved by the design should be able to be evidenced through the notation. The use-case diagram is a start but needs to be linked back into the actual design – or else it is just mimicry.

Integrating static structures and dynamic behaviour is difficult. Students must be able to discern the components of the system, while also seeing how they work together. The skill to do this takes time and practice. While students cannot achieve expert status, they can be given the opportunity to design systems of increasing complexity as they progress through the curriculum.

Conclusions

'Threshold skills' have been proposed as a complement to threshold concepts (Thomas et al. 2012, Sanders et al. 2012). In this paper we have investigated software design as a possible threshold skill. We have seen that it displays many of the same characteristics as a threshold concept, albeit with the slightly different focus that we described as a threshold skill. It is troublesome, integrative, semi-irreversible and must be practiced.

We used interviews which sought out threshold concepts and then extracted information on software design and skills. Furthermore, we asked a group of graduating computing students to 'design a super-alarm clock that University students could use to manage their sleep patterns'. By examining our students' designs, we were able to see both skill – what they do when asked to design - and something of what they understand of the concept of design.

It appears that software design is a threshold skill in the terms we considered. Overall, increased design skill is both transformative (it changes the students' understandings of what they can do) and integrative (the design for one program can be adapted and re-used for another). In terms of reversibility, there may be an issue with granularity: parts of the design process – fluency in computing language and software design notation and building a solution from those parts is semi-reversible and may need refreshing. But the ability to break down a problem into parts may be irreversible. From the comparison of the results of this study with other studies that used less prepared students, students improve with practice; yet software design continues to be troublesome and to serve as a boundary marker in the computing field. While learning software design, students exhibit characteristics of being in liminal space.

At the TC conference in Durham the point was made that it is not enough to *discover* threshold concepts – that is only useful if it leads to developments in learning and teaching. By conducting this analysis we have been led to consider what were the critical parts of completing a software design and what are the implications for teaching this difficult subject.

References

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007). Threshold concepts in computer science: do they exist and are they useful? In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education,*7-11 March 2007*,* pp. 504-508. Covington, KY: USA

Box, I. (2009).Toward an understanding of the variation in approaches to analysis and design. *Computer Science Education*, *19*(2), 93–109.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., & Zander, C. (2006). Can graduating students design software systems? In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education,* 1-5 March 2006*,* pp. 403-407. Houston,TX: USA.

Flanagan, M.T. (2012). *Threshold Concepts: Undergraduate Teaching, Postgraduate Training and Professional Development: A short introduction and bibliography.* Retrieved from http://www.ee.ucl.ac.uk/~mflanaga/thresholds.html.

Loftus, C., Thomas, L., & Zander, C. (2011). Can graduating students design: revisited. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education,* 9-12 March 2011, pp. 105-110. Dallas, TX: USA.

McCartney, R., Boustedt, J., Eckerdal, A., Moström, J. E., Sanders, K., Thomas, L., & Zander, C. (2009). Liminal spaces and learning computing, *European Journal of Engineering Education*, *34*(4), 383-391.

Norman, D. A. (1990). *The Design of Everyday Things.* Doubleday, New York.

Norvig, P. (2001). *Teach Yourself Programming in Ten Years.* Retrieved from http://norvig.com/21-days.htm, September 28, 2014.

Oxford English Dictionary Online. (2016). Oxford University Press. http://www.oed.com/

Rountree, J., & Rountree, N. (2009). Issues Regarding Threshold Concepts in Computer Science. In *Proceedings of the Eleventh Australasian Conference on Computing Education, (ACE '09),* 19-23 January 2009*,* pp. 139-145. Wellington: New Zealand.

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Thomas, L,. & Zander, C. (2012). Threshold Concepts and Threshold Skills in Computing. In *Proceedings of the ninth annual international conference on international computing education research (ICER '12),* 9-11 September 2012, pp. 23-30. Auckland, New Zealand.

Shinners-Kennedy, D. (2008). The Everydayness of Threshold Concepts: State as an Example from Computer Science. In R. Land, J. H. F. Meyer, & J. Smith (Eds.), *Threshold Concepts within the Disciplines*, pp. 119-128. Sense Publishers, Rotterdam.

Sorva, J. 2010. Reflections on threshold concepts in computer programming and beyond. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research *(Koli Calling '10),* 28-31 October 2010, pp. 21-30. Koli: Finland.

Thomas, L., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., & Zander, C. (2012). A Broader Threshold: including skills as well as concepts in computing education. In *Threshold Concepts: from personal practice to communities of practice. Proceedings of the National Academy's Sixth Annual Conference and the Fourth Biennial Threshold Concepts Conference*, 27-29 June 2012. Trinity College, Dublin: Ireland.

Thomas, L., Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., & Zander, C. (2014). Graduating Students' Designs – Through a Phenomenographic Lens. In *Proceedings of the tenth annual conference on international computing education research (ICER'14)*, 11-13 August 2014, pp.91-98. Glasgow: U.K.

Zander, C., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Sanders, K. (2008). Threshold concepts in computer science: a multi-national investigation. In R. Land, J. H. F. Meyer, & J. Smith (Eds.), *Threshold Concepts within the Disciplines*, pp. 105-118. Rotterdam: Sense Publishers